CHAPTER 1

Enterprise Applications

From the beginning of time, people have been trying to make the quality of their lives better by finding new ways to do more tasks in less time with less effort. This is evident by the technological advances humans have made such as fire, the wheel, the telephone, and the computer. As each discovery is made, we then improve on that discovery to make it more efficient and cheaper to produce. This desire to make our quality of lives better is what drives us—it makes us who we are.

This desire is quite clear in the world of software development, where we continually try to discover better ways to create software so that it runs faster, is developed faster, is cheaper to produce, and can do more. One of the key strategies that the software development community discovered early on was the reuse of software code. Unfortunately, in many places, the same software code would be “copy and pasted” and used repeatedly within an application, making development and maintenance tasks a real challenge.

As time has gone on, many improvements, such as reusable methods, object orientation, service orientation, and so on, have been incorporated into the process of developing software. However, these software development techniques alone will not make software development easier. These are just the building blocks that you must build upon to ensure an ideal project outcome.

Understanding the architecture and needs of a software application will give the developer a better understanding of how the Enterprise Library can assist in developing an application framework. This framework could be the basic foundation of services for all applications within a particular organization. In this chapter, I’ll cover the basic fundamental building blocks of today’s software applications. The rest of the book will then show how the Enterprise Library can help ease the development of these application building blocks.

The Needs of a Software Application

Successful software development is possible only through proper planning. Without doing so, you may find the application took longer to build, wasn’t what you or the user expected, or worse yet was almost impossible to maintain. Proper preparation and design are integral to the creation of any software application regardless of its size and complexity.

When creating a new application, you first need to understand the type of business the application is for or in some cases who the audience of the application is if it is not specific to a business. Understanding the needs of the stakeholders and users is critical to the success of any application being built. For instance, a money management company responsible for securities trading would require applications to be very responsive because actions must be taken on a timely basis once a buy or sell decision has been made. However, if your application is tasked with working with batch processes, performance may not be as important as scalability. These two examples indicate that the audience will drive the design of the application indirectly based on what its needs are.

The end user's purpose will help define the general technical and design needs of an application; however, gathering the specific requirements from the business will determine the application's design requirements. From these business requirements, you should create a set of functional requirements detailing the specific functions required of the application. This is probably a good place to create use cases to define the actions and tasks that can be performed in the application. Finally, you can create the technical requirements for the application. This is where you get to create all the fun UML class diagrams and sequence diagrams as well as the data modeling. However, before you get to the technical specifications, you should consider some issues first.

The first elements to understand are the growth and current direction of the business. Is the business looking to expand its product line? Has the company been trending toward a 30 percent growth rate over the past five years, or is growth more along the lines of 300 percent? These types of questions are important to ask and understand, because it will help determine to what degree the application should scale. Although I'm aware that some professionals think you should never design an application beyond meeting the business and functional requirements, I think you should consider exposing certain interfaces so that future expansion can be done with less effort. However, such considerations should be balanced with the understanding that an application will not be able to account for every possible expansion scenario.

During the application design phase, you'll soon discover that you will be repeatedly implementing many of the same features and functions. The following are common examples of this recurring code:

- Retrieving, inserting, updating and deleting database data
- Logging application events
- Handling application exceptions to ensure that they are properly escalated
- Retrieving and updating application settings
- Improving performance by providing a caching mechanism to reuse static data performance

You can implement these common functions in a few ways, discussed in the next sections.

Copy and Paste

One way to implement these examples is to copy and paste code throughout the application. Granted, this may seem extremely easy, but maintainability is nearly impossible. Think about an application that requires access to a database. Typically the creation of the connection object would be the same for all retrieve, insert, update, and delete functions. Assuming there are three tables used for this application, it would be safe to say there would be about 12 instances where a connection object would be needed. So, copying and pasting 12 times seems relatively easy. But if a change is required for one of the three tables to point to another database, you now have to go into the application and find the appropriate locations to change and modify them. To make matters worse, what if it became necessary to change the ADO.NET data provider? You would need to find all the connection object creation code and modify it. In short, although this approach results in short-term gratification, the long-term effects can be error prone and costly.

Code Generation

Another way to handle this repetitive code is by creating it via a code generator. Code generators are very good at creating a lot of code quickly. But although they are adept at creating stable, bug-free code, code generators aren't without their own issues. First, modifying the generated code will prevent the developer from being able to regenerate the code using the code generator, since the code generator does not have the ability to incorporate the changes. This will effectively make development no easier than the copy-and-paste scenario described earlier. Second, they are applicable only given a certain set of conditions. If the code generator is unable to address a specific requirement, then it will be necessary to modify the code generator to address that need. Another issue with code generators is that typically all the code generated can be easily modified by the developer, and this can be problematic in environments where a specific technique is desired to handle a function such as creating a connection object. This is not to say that using a code generator is bad, and in fact, used properly, it can be a useful tool during the development of an application or component and can be used alongside other techniques of handling repetitive code such as frameworks.

Frameworks

Another common technique is to create a series of components that can be reused throughout an applicationto perform the desired common functionality. These components can typically be used as is or subclassed to provide special implementations. The use of these components allows for cleaner, simplified code, which in turn allows for maximum maintainability while providing simplified interfaces for the developer to use. Together, these components create a foundation that an application can be built upon. Another common name for this collection of components is a *framework*.

Common Framework Types

An application can utilize one of two types of frameworks: an *environment framework* (sometimes referred to as a *development and execution environment framework* or *system framework*) and an *enterprise framework* (also known as an *application framework*). These two frameworks work together to provide the services and interfaces necessary to develop software applications. I'll now introduce you to both types.

The Environment Framework

Every mainstream operating system has a set of application programming interfaces (APIs) that expose its features and functionality. Generally, this API is quite robust and exposes interfaces that most applications may never use. Additionally, to utilize the features of the operating system API, a fair amount of repetitive code must be written in order to perform simple tasks such as creating a window. This is where an environment framework can simplify the utilization of system resources.

The environment framework typically provides a set of interfaces and features available within the operating system environment to a development environment. These interfaces and features hide all the necessary environment-specific code to implement specific tasks. Some examples of execution environment frameworks include the Java Virtual Machine, Visual Basic runtime, and the .NET Framework. These frameworks provide reusable code that makes tasks such as creating a window simple. With the environment frameworks, you typically do not have to worry about low-level tasks such as memory allocation, file handlers, windows handles, and so forth.

The Enterprise Framework

It is great that we can take advantage of environment frameworks when developing applications, but most applications are going to have greater framework needs. Granted, when you are developing a Windows or web application, the environment framework can save you from disaster by giving you a friendly and relatively safe environment in which to create your masterpiece. However, if you are not careful, you will find yourself with unmanageable code that is hard to read.

The problem is that most development environments such as the .NET Framework provide more than one way to perform a specific task. Suppose a business is going to require a new reporting module that contains multiple reports, and three developers—Steve, Andrew, and Sue—have been assigned to create this reporting module. Since the three reporting module contains three separate reports, the three developers decide to develop one report each.

Steve determines that he needs to retrieve a table of lookup values that will be used to determine the proper criteria to create the report. Therefore, the first task Steve decides to do is create a data access class. And in this new class Steve creates a method called `GetReportLookupData` to retrieve a drop-down list of data needed for a report:

```
public class myLookupDataAccessClass
{
    public ReportLookUpData GetReportLookUpData()
    {
        //Some code...
    }
}
```

At first glance, it seems like a pretty simple implementation. Steve will write some data access code within the `GetReportLookUpData` method that is going to return a custom business entity called `ReportLookUpData`. Therefore, Steve decides to use a `SqlReader` to retrieve the lookup data. To create the `SqlReader` object, Steve must create a `SqlCommand` object to execute the SQL statement and a corresponding `SqlConnection` object to connect to the database. Now the `SqlConnection` object requires a connection string to determine specific parameters required by SQL Server in order to connect to it. In this particular case, Steve decides to just hard-code the string into the code:

```
public class myLookupDataAccessClass
{
    public ReportLookUpData GetReportLookUpData()
    {
        string myConn = "server=MyServer;database=MyDatabase;Integrated
```

```
                                Security=SSPI";
        ReportLookUpData lookUp = new ReportLookUpData();
        SqlConnection connection =  new SqlConnection(myConn);

        //Data retrieval code...
        return lookUp;
    }
}
```

While Steve was creating his data access component, Andrew was working on a data access component for another report. Andrew's experience told him that hard-coding the connection string would not be the best choice, so he decided to use the application's configuration file to store the connection string.

Finally, Sue was working on yet another data access component for the report she was creating, and she decided to store the connection string in the Windows registry.

From the developers' standpoint, the application is ready to be deployed in the production environment, with the connection strings hard-coded, stored in the Windows registry, and stored
in the application configuration files.

A few months pass, and a new developer named Tim joins the company. Tim is tasked with changing database providers from SQL Server to Oracle. Tim immediately sees an issue with Steve's hard-coding of the connection string, so he opts to use a text file to store the connection string for the application.

Now Tim not only has to modify each data access component but he also must discover how the connection strings were stored and modify all the storage mechanisms in order to change the connection string between the development, integration, and production environments. For the very smallest of applications, this may not be a big deal. However, for very large applications, where there may be hundreds of data access components that require connection strings, this can become a maintenance nightmare.

Having a common, consistent way of handling connection strings is most beneficial from both the development and administrative points of view. To accomplish this, an organization might define a coding standard defining how to properly store and retrieve a database connection string. However, this technique will end up creating many coding rules that the development team will constantly have to remember and enforce. Even with a coding rule like this in place, the application developers would still have to create the code to retrieve the connection string, and this repetitive coding would eventually become mundane and time-consuming.

A better way to handle these kinds of development issues is through an enterprise framework. An enterprise framework builds upon the functionality available within a development environment, but it takes the most common code routines in an application and encapsulates the code into a more simplified implementation. Let's reconsider the previous example, this time revising it to use a framework component:

```
public class myLookupDataAccessClass
{
    public ReportLookUpData GetReportLookUpData()
    {
        ReportLookUpData lookUp = new ReportLookUpData();
        SqlConnection connection =  new
            SqlConnection(AcmeFramework.GetConnectionString());

        //Data retrieval code...
        return lookUp;
    }
}
```

Now Steve has used the `GetConnectionString` method in the AcmeFramework component to retrieve the connection string. Steve does not have to worry about the details of how the connection string is stored or how to retrieve it. The AcmeFramework will handle the details for him.

Now when the application is promoted to production, the production enterprise framework will handle the details of how to retrieve the connection string. Hence, it is not required to modify the application configuration file or Windows registry to add the new database connection string. In addition, the developers will typically be required to write fewer lines of code using an enterprise framework as opposed to writing a custom implementation.

Another benefit to using an enterprise framework is the ability to change the underlying implementation of the framework without having to always touch the public interfaces it exposes. For example, let's assume the AcmeFramework utilizes an XML file to store configuration data; however, company policies change, requiring all connection strings to be encrypted. Instead of having to change every data access component's connection string implementation, only the `GetConnectionString` method is concerned with decrypting the encrypted connection string inside the XML file.

Overall, an enterprise framework will help enforce consistent software development, require less code, require fewer bugs, and provide the use of best-practices implementations within an application. Other benefits of using an enterprise framework include freeing developers from having to code low-level tasks, such as opening and closing files, and allowing less experienced developers to develop an application. From a project management standpoint, this can lower a project's development costs and at times allow developers who have more business than technical skills to play a bigger role in the development of an application. In the end, an enterprise framework is not a silver bullet; it should provide the core common functionality and features needed for most, if not all, of your software application as well as allow for more project success stories by letting the developers focus on implementing the application's requirements as opposed to worrying about connection string management.

Core Components of an Enterprise Framework

To understand the components of an enterprise framework, you first have to understand the components used to create an application. Each application has to perform certain functions in order to meet the needs of the user. These functions can range from accessing data to sending email messages to formatting a document. To perform these tasks, it makes sense to try to break them out into specific components. This breaking apart of the features allows for easier maintenance of the application, as well as the ability to scale out as needed.

Most business applications require some sort of user interface to interact with data, a mechanism to validate data entered in by the user, and the ability to read and write data. Typically, the components are called *separated tiers* or *layers*. I prefer the term *layers* when thinking of the logical separation of components and *tiers* when referring to the physical separation of components. The typical high-level layers of an application are the presentation layer, the business logic layer, and the data layer. However, the needs of an application do not stop with these components of an application. As previously mentioned, an application may also need components to handle security, application configuration, exception handling, logging, application deployment, and so forth. The specific application components will be determined based on the functional and technical requirements of the application. However, as an enterprise grows and applications are developed, it will soon become apparent that all these applications share common functionality between them. This common functionality is the perfect candidate for common enterprise framework components. Figure 1-1 shows the different components that are typically used in an application. In the following sections, I'll introduce several of the most common components.

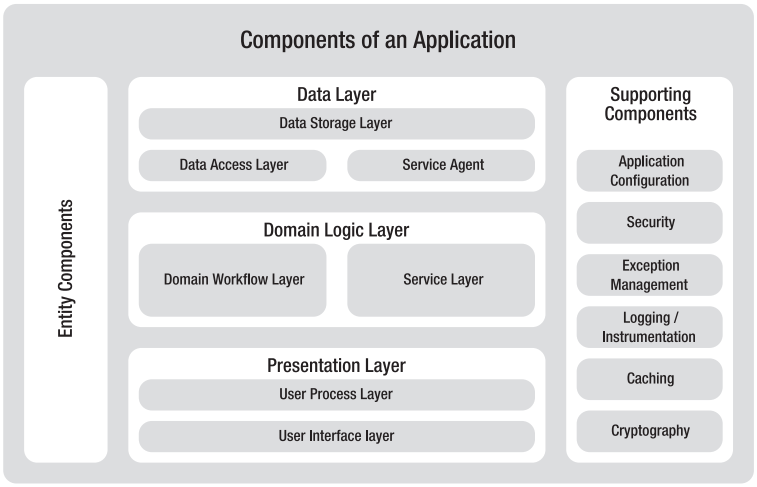

Figure 1-1. *Components of an application*

Data Layer

The *data layer* can be broken into three sublayers in most applications. They are the data storage layer, the data access layer, and the service agent, as shown in Figure 1-1. The data storage layer provides the mechanism for storing and managing your application data, and the data access layer provides the logic needed to retrieve and manipulate data. The service agent is like the data access layer, but instead of accessing a data storage layer, the service will access another domain process such as a web service or COM+ component via DCOM. Together these three logical layers provide the mechanisms for gathering and manipulating data.

Data Storage Layer

Typically, the data storage layer consists of a relational database server such as Microsoft SQL Server 2005 or Oracle. This layer can be shared between multiple applications, but there should be some logical or physical separation within the database server. One example would be having a database for each application that the database server is supporting.

The data storage layer provides a series of functionality to an application such as retrieving, inserting, updating, and deleting data. Not all data storage layers are relational database servers. An XML document utilizing an XML parser such as the MSXML DOM or a SAX parser could also be considered a data storage layer. Some other examples of data storage layers could be the Windows registry, a .NET application configuration file, or even a Microsoft Excel spreadsheet. However, relational databases are the most common for storing application data. They typically abstract the methods of "how" to retrieve data and instead expose functionality on "what" data to retrieve, which lets the database server figure out the "how." The most common language used for retrieving data is SQL or some derivative of it.

The data storage layer is typically called only by the data access layer; by reporting and analysis tools; or by other extraction, transformation, and loading (ETL) applications. Hence, you should not interact with the data storage layer from the user interface or business layers; doing this can result in an application that does not scale well or that cannot be easily maintained.

Data Access Layer

Ordinarily, the *data access layer* consists of one to many classes or components. These classes handle all the work necessary to read and manipulate data. They provide a consistent abstract interface to the data storage layer, so the rest of the application does not have to worry about how to get at the data. The data can reside anywhere from a simple text file to a relational database such as SQL Server 2005. The data access layer will typically be consumed by a business logic layer, sometimes by the user interface layer for retrieving lookup data for drop-downs controls, or by reporting engines.

It is important to know that the data access layer should at least be structured logically. In other words, it does not have to consist of just one class or assembly, but at the same time, it should consist of no less than one class within the executing application assembly. A common way of logically structuring the data access layer is to have one class dedicated to a logical group of data. An example of this would be having a `customers` class that is directly related to a group of `customer` tables. The decision of whether you want to have your data access logic inside the main executing assembly or physically separated like in an *n*-tier application will be based on the scalability, maintainability, and performance needs of your application. By containing your data access logic with one or more classes, you will gain the advantage of being able to swap out a data access component with another one. For instance, suppose that the application your organization has been using currently is utilizing an Oracle database. Now with .NET 3.0 and SQL Server 2005 being the latest and greatest development technologies, the powers that be have made an executive decision to migrate all databases to SQL Server 2005. For the most part, all you would have to do is create a new class that now utilizes the SQL Server client provider (`System.Data.SqlClient`) as opposed to the Oracle provider. Then test the new component, and put it in production. From a high-level point of view, not all implementations may be as simple, especially when using SQL commands that are specific to a database provider.

The key to allowing this is that the public interfaces that were exposed by the Oracle data access component should match the new public interfaces exposed by the SQL Server data access component. Testing should be just as simple, and your original unit tests should work with the new data access component just as it did with the old data access component. Again, this is facilitated by the fact that the public interfaces did not change. Or at least you should not have to change the interfaces to swap out database providers. The following is an example of how two data access classes can each have the same interfaces yet each utilize a different database for its data:

```
public class myOracleDataAccess
{
   public DataSet GetSalesReport(DateTime beginDate, DateTime endDate)
   {
      DataSet myDataSet = new DataSet();
      string myConnString = "Data Source=Oracle8iDb;Integrated Security=yes";
      OracleConnection myDbConn = new OracleConnection(myConnString);
      OracleCommand myDbCmd = new OracleCommand();
      myDbCmd.Connection = myDbConn;

      //Oracle Data Adapter, command string, and parameters added here

      myDataAdapter.Fill(myDataSet);
      return myDataSet;
      }
}

public class mySqlDataAccess
{
   public DataSet GetSalesReport(DateTime beginDate, DateTime endDate)
   {
      DataSet myDataSet = new DataSet();
      string myConnString =
         "Server=SqlServerDb; Database=Northwind; Trusted_Connection=true";

      SqlConnection myDbConn = new SqlConnection(myConnString);
      SqlCommand myDbCmd = new SqlCommand();
      myDbCmd.Connection = myDbConn;

      //SQL Data Adapter, command string, and parameters added here

      myDataAdapter.Fill(myDataSet);
      return myDataSet;
   }
}
```

Examining the two classes, `myOracleDataAccess` and `mySqlDataAccess`, you'll notice each one has a method called `GetSalesReport` that returns a dataset. You will notice in both instances that the method signatures are the same; hence, you could pull out the Oracle data access class and replace it with the SQL data access class. You, of course, would want to make sure that the results returned are identical between the two classes for the given method.

Service Agents

Service agents, also referred to as *proxies*, access information from another domain process. In very large organizations, it is beneficial to keep different applications loosely coupled. Thus, each application can utilize the best platform and technology to perform the necessary functions that it requires and yet provide interfaces to other applications to consume these functions. Service agents can also be used to retrieve data from a third-party vendor. An example of this is a retail application requesting a shipping cost from a courier by passing destination and pick-up ZIP codes to the courier, with the courier responding with a shipping cost.

Service agents are not a new concept; they have been around for years—from the crudest of interfaces such as passing an ASCII file from one server to the next to more elegant solutions such as DCOM and today's web services.

■ **Note** I prefer calling business logic *domain logic* because the word *business* implies the logic is for business purposes, and I prefer a more generic term to encompass any kind of problem or process that requires rule validation—business or not.

Domain Logic Layer

The *domain logic layer* (more commonly referred to as the *business logic layer*) is where you will process all the rules and validation needed to perform an action. This layer, like the data access layer, should at least be its own class to allow for easy replacement if necessary, but it does not have to be a separate assembly.

■ **Note** I prefer the term *domain layer* because the term *business layer* should always refer to "business" rules. The word *business* is too narrow of a concept in the vast world of application development. A perfect example of this is an Internet email client validating that a user has entered a proper email address; this task is important, but it's not specific to a business purpose.

The purpose of the domain logic layer is to mimic what was once performed by another domain process whether it was done automatically in a software application or done manually by a person. For instance, in a retail point of sales (POS) process, a clerk would have to manually write each item being purchased on a sheet, then tally the total of all the items, and finally add any sales tax. In a POS application, listing the items being purchased, tallying them, and adding sales tax is all done by the application. The location within an application where this is done is typically the domain logic layer.

By keeping the domain logic together in one layer of your application, you are also going to simplify its maintenance. The domain logic layer typically sits in between the data access layer and the presentation layer. Hence, as a user enters data into the application, the data is validated against the domain logic and then passed to the data access layer to be saved into the database.

The domain logic layer, like the data access layer, should at least be a logical separation within your application. However, this does not prevent you from having multiple classes, assemblies, and so on, in your domain logic layer. Also keep in mind that not all domain logic components will necessarily communicate with a data access layer. Some of these components will be stand-alone such as financial math calculators; some typical names for these kinds of domain logic components are *common logic components* or *common domain logic components*.

Finally, when it comes to the domain logic layer, remember that not all of your domain logic will be able to reside solely in this layer. Yes, it is important to strive to get as much as your domain logic in one manageable location, but sometimes this is not practical. One good example of this is when performing business-specific calculations. It may make sense to perform these calculations within the domain logic layer, but it may also be practical to perform the same calculations while doing a bulk upload of data into the database. Therefore, you may have one component deployed to both the database layer on the database server and the business layer on either an application server in an *n*-tier environment or the workstation in a client/server environment. Again, you will have to balance out maintainability, scalability, and performance requirements based on your application needs.

Domain Workflow Layer

This layer, a subcomponent of the domain layer, handles the processing of workflows. Typically, the components built in the domain layer should be very specific to a domain problem. Some examples of this are adding a new customer, adding a new order, requesting shipping costs, and calculating sales tax. The domain workflow layer would handle the process of creating a new order by orchestrating the domain logic calls and transactions.

The domain workflow layer is not for every application. Its use should be determined based on the application needs. For instance, a simple maintenance application that maintains a list of countries would probably not use a domain workflow layer. However, an application for an investment company may use a domain workflow to manage the process of trading securities.

Service Layer

The *service layer*, also known as a *façade layer*, provides an entry-point mechanism for other applications to access specific domain functions or services of an application. It allows you to provide a black-box interface to your application so that the caller doesn't need to know the internal details of domain logic. Typically, service agents consume service layers, and some common implementations of service layers are web services, CORBA, and COM+. Service layers typically will perform necessary data mapping and transformations as well as handle processes and policies for communicating between the domain layer and consumer.

Interoperability between heterogeneous systems is not a requirement for a service layer; it is perfectly acceptable to support just one platform in your service layer implementation. Interoperability can introduce performance issues and system limitations. The need to provide an interoperable service layer will be based on the overall requirements of the domain. For third-party vendors, it is probably a great idea to utilize web services for its service layer.

For an internal application where all the applications are on one platform, utilizing .NET remoting or COM+ via DCOM might be a better solution.

Note It is important to note that the service layer doesn't necessarily imply the use of web services. Web services are just one implementation of service layers; other implementations might use .NET remoting, DCOM, CORBA, and so on.

Presentation Layer

The *presentation layer* typically consists of one or two sublayers, namely, the *user interface layer* and the *user process layer*. In most smaller applications, it is necessary to have only the user interface layer. However, in large applications or applications with multiple types of user interfaces, a user process layer would prove beneficial. The user process layer would handle the common user interface processes such as wizards and the interfaces to the domain logic layer.

Like the data access layer, you will sometimes have to keep some logic in the presentation layer. However, this domain logic is very basic and is typically used for validating data types and formatting. A few examples of this would be validating that a phone is formatted correctly or that an entered credit card number contains only numbers.

Also keep in mind that it is fine to call a data access layer directly from the presentation layer; however, this should be done only for retrieving lookup values in a combo box or list box or for reporting purposes. All data manipulation should be done strictly through a domain layer. You also have to keep in mind that calling the data access layer from the presentation layer reduces your application's scalability.

User Interface Layer

Most applications are designed with the intention that a user will interact with it. The user interface layer will contain all the user interface components such as web or Windows forms. These user interface components are then used to interact and communicate with the domain logic layer and sometimes the data access layer.

An important thing to remember about the user interface layer is that you should keep domain logic to a minimum. If you are using a user process layer in your application, you should have practically no domain logic whatsoever in the user interface. Any domain logic should then be part of the user process layer. The one exception to this rule is a web application; for performance and usability reasons, it may also be necessary to apply some domain logic in the HTML page as client script.

Tip In web applications, it is important to remember that even if some domain logic is being performed in the browser, you still have to perform it on the server to ensure the domain logic is applied. Not all environments can guarantee that the web browser has scripting turned on. This is very true for business-to-consumer applications.

User Process Layer

With larger applications where you have rich, robust user interfaces or many types of user interfaces, it may become more critical to manage the processes or workflows of the user interface in a separate layer. This allows the abstraction of user interface components from the actual process that a user must undertake to complete a given task. The user process layer would also manage state and calls to the domain logic and data access components. Using a user process layer will help make your user interfaces very lightweight and ideally give you the ability to easily create multiple types of user interfaces without having to do much more than create your Windows or web form and drop some UI controls onto it.

The Model-View-Controller (MVC) design pattern is a good implementation of a user process layer. The model would manage the state and calls to the domain logic components. The view would be the user interface components themselves. Lastly, the controller would handle events, handle workflows, and make the necessary calls to the view and model. In this case, the model and controller are the components of the user process layer, and the view is the component of the user interface layer.

Entity Components

An *entity component*, also referred to as a *business entity*, should represent an entity within a domain. A customer, sale item, employee, and sales transaction are all typical examples of an entity. Each one of these entities can be represented as an entity object. The entity component will contain all the attributes necessary to perform the tasks that it is related with. The entity component is typically shared between all the layers of an application, because the entity component is the primary way you would pass the application data around your application.

For example, an entity component that represents an employee in a retail application may contain the following attributes: first name, last name, Social Security number, employee number, and home address. The Social Security number, last name, first name, and address attributes are required for printing the employee's paycheck. The first name, last name, and employee number attributes are required during a sales transaction. In this case, one entity component can be used for sales transactions and employee payroll. However, sometimes when an entity has many attributes, these attributes are specific to certain domain tasks. It may be necessary to create more than one entity component to represent a domain entity.

One way to minimize the amount of redundant code is to use inheritance when designing your entity component. In this case, you would build a base component called `person`, and a person would have a first name, last name, and address. The inherited class would contain all the attributes the base class has plus any new attributes it would add. Since a customer and an employee both require a first name, last name, and address, you would inherit from the `person` base class and create a `customer` class and an `employee` class. The `customer` and `employee` classes can then add specific attributes for a customer or an employee. Therefore, a `customer` entity might add a preferred shipping method attribute and a birth date attribute. The `employee` entity might add a Social Security number attribute and employee number attribute.

Also, in some architectures, an entity component can be part of the domain layer. An example of this is in an object-oriented architecture; the entity object would also contain the necessary methods for performing data manipulation upon itself. Although this kind of implementation would be considered a good OO design, in some cases scalability and performance may be sacrificed while taking this approach. This is why most applications take a component-oriented architecture or service-oriented architecture approach and pass the entity component to a domain component where some action is taken on that entity component.

Application Configuration Data

Every application needs to contain metadata that will define the application's execution environment. Some examples of metadata include a database connection string, FTP server addresses, file paths, and even sometimes branding information. To provide a way to set this configuration data in an application, most applications depend upon an INI or XML file to store that data. With .NET applications, it is easy to utilize the application configuration file to store your configuration data in an XML format. You can utilize the built-in `<appSettings>` element setting configuration settings, or for more complex scenarios where you have complex hierarchies of configuration data, you can create your own custom configuration section.

Some of the downsides of using the .NET application configuration file are that the files are read-only at runtime and it's not possible to centralize data between multiple applications. These limitations may force larger applications to come up with a custom solution to store the configuration data for an application. Also, currently it is not a good user interface for an administrator to configure the application configuration file. This can make administrating this file difficult and cumbersome when attempting to read and modify these files with a large amount of configuration data.

Some other options you can look at to store configuration data are the Windows registry, a file stored locally, or a file stored on a network file server; you can even use a database server to store application configuration data. The key thing you want to remember is to determine the features of the configuration data needs based on the current application requirements and the potential growth of the application.

Managing Security

Another important application need is securing the data and features that an application provides to its users. To do this, an application must identify and then determine what data and application rights it can access. Another set of terms for this is *authentication* and *authorization*. Some of the challenges faced with application design are determining a simple way of managing security between the different layers and determining the different types of user interfaces that may be required for the application.

Another challenge is also determining what is the best way to implement the security management of an application. Some things to consider in this decision process are as follows:

- Is the application in-house, or are you a vendor building this application for your clients?
- How will the application be accessed? Will it be strictly internal, or will it be accessible via an extranet or over the Internet?
- What portions of the application will be exposed to whom? Will it be necessary to ensure that the sales group cannot access the executive manager's reports?
- Does the application have to worry about being platform or version independent?
- Do the security mechanisms have to be shared between heterogeneous applications?

Once you have determined the needs of your application, you can determine the best approaches for securing your application.

Authentication

The first step you must perform to secure your application is to determine the identity of the person or system that is trying to access it. For .NET applications, you have two basic choices: you can authenticate either with Windows authentication or with non-Windows authentication. Both of these have their pros and cons.

When utilizing Windows authentication via Active Directory, you are allowing the operating system and domain to determine the identity of the person or system trying to access it. This usually takes place by a user or system supplying a username, password, and domain to the application. Once those are supplied, the application will call upon the operating system to verify the credentials presented. In a Windows application, this is done just by the fact the user has logged onto their desktop. A Windows service provides the credentials supplied to it to the operating system. For a web application, an Internet browser will attempt to utilize the credentials that the user is currently running the web browser with. However, if the web application server cannot verify the credentials, then the web server may give the user an opportunity to supply the correct credentials to access the web application. In the case of a web service application, the application calling the web service needs to have the credentials supplied. Depending on the design of the web service proxy component, this can be defaulted to the credentials that the user is logged in as, or another set of credentials can be supplied by the application to the web service.

In a non–Active Directory authentication scenario, the burden of verifying a user is put on the application. In Windows applications and Windows services, it is up to the application to look up the credentials provided and verify them against a custom implementation. One example might be taking a username and password and validating against a database table of usernames and passwords.

In web applications, you have a few more choices in how you can validate a user. You can do it manually like a Windows application, but then you are required to put in this authentication code for each web page of your application. This is to prevent someone from gaining access to your application by attempting to navigate to some web page other than your intended main entry or login page. Or a better solution would be to use ASP.NET forms authentication. Forms authentication takes users who are not validated and redirects them to a login page where they can supply their user credentials. Once authenticated, they are free to navigate the web application as they like. Forms authentication utilizes cookies to determine whether the user is known or unknown as they navigate the application. The credentials can be stored in the web application configuration file or can be a custom implementation such as the custom database scenario described for the Windows application.

In the scenario of a web service, the same issues that existed for the web application also exist for a web service application. However, they are harder to resolve. In a web application, forms authentication would redirect the user to a login page. In a web service, that is not practical, so it will be necessary to authenticate the user before taking any other action. This will require the application consuming the web service to call a login web method where the web service can authenticate and issue a cookie to the calling application. What makes matters more difficult is if the calling application is a web application, you have to manage state to retain the authentication cookie. You can do this by storing the cookies in a session variable. In all web services, you probably want to steer away from forms authentication. The good news is there are other technologies such as Web Services Enhancements (WSE) that specifically address security issues for web services.

Authorization

Once you have authenticated a user or system process, the next task is to determine what features and functions they have access to in your application. The first choice you have to make is whether you want your application to authorize access on a per-user basis or whether you prefer to assign the user to a group and grant the group access to specific features.

In the scenario where you assign access to features and functions in your application on a per-user basis, you will find for larger applications that administration will soon become a nightmare. However, for very small applications, authorizing users directly might be acceptable. You will have to determine this during the design of your application.

Assigning groups of users to a specific feature or function, better known as *role-based authorization*, will prove beneficial in moderate to large applications. Once again, you have two high-level choices you can choose from when implementing role-based authorization: either the operating system can help manage it or you can build your own custom solution.

In allowing the operating system to help manage role-based authorization, you will probably be using Active Directory to manage your groups. Thus, you will assign an Active Directory group to a specific feature or function. Then you will assign users to active directory groups. When a user authenticates to the operating system, you can then determine which active groups the user belongs to and determine authorization to the features and functions of your application based on those active groups.

Some key points to remember when using Active Directory are as follows:

- You can use Active Directory only if the user is authenticated to the operating system and domain.
- When dealing with applications that are intended for public availability such as web applications, performance and maintenance of Active Directory may become an issue.
- Active Directory does not interoperate well with other heterogeneous systems like Unix servers.

Another approach to handling role-based authorization is to create a custom implementation. One example like the custom authentication scheme mentioned earlier is to utilize a database to store groups of users. This way you can have user credentials related to user groups and then have user groups related to application features and functions. This approach offers more flexibility than Active Directory in that it can be implemented for different operating system environments. It can use the operating system to authenticate a user while still using this custom implementation. Finally, a database will typically perform better than Active Directory, especially with large volumes of users and groups like in the scenario of a public web application.

The downside is that you have to implement this beforehand, so you will need to create the data structures to store the data. You also will probably want to implement a maintenance application to maintain the users and groups. Overall, like everything else, the requirements of the application will determine the best approach.

Handling Exceptions

As much as we like to believe that we create bug-free applications, this is almost certainly an unrealistic aspiration. Sometimes it is because of a bug within a technology we are utilizing, but most of the times the exception will occur because we introduced a bug or error into the application. Handling exceptions is a critical task in an application; it provides a way for the application to inform the user when something goes wrong.

Out of the box, .NET provides a mechanism for reporting exceptions to the user; unfortunately, the messages are not user-friendly. On top of that, when a user sees an error, 90 percent of the time they will close the application and try to restart it. If the function they were trying to perform works after the restart, more than likely you will never know the exception occurred. Granted, some individuals believe that ignorance is bliss, but in this case an unhandled defect in an application can reduce confidence in the application as well as in the developer who wrote it.

This means as developers we have to anticipate the errors that can occur and try to gracefully recover from them. Sometimes you can do this without telling the user anything and simply log it to an exception log file. Other times, you have to notify the user and perform some kind of action such as canceling the task they are trying to perform or closing the application. When you are handling specific errors, you can then not only notify the user of the issue but also notify the developers via an exception log or possibly an email. This allows the developer to be aware of any issues regardless of whether the user opted to notify the developer.

In addition, we can't anticipate certain errors, such as the users attempting to use the application it was never intended for. These unanticipated exceptions should be handled globally. In this case, you may have to close the application, but you can still notify the user with a friendly message, as well as send a message to the support staff. This can give the support staff the opportunity to address the issue before it becomes widespread.

In either case, handling errors in your application will help you improve the quality of your application as well as save face in front of your users. Also, handling errors gives you the opportunity to cleanly close down an application or feature of an application, thus reducing the possibility of memory leaks.

Logging

Along with handling exceptions, it can also be beneficial to log events within an application. Logging can help determine the application's performance, create audit trails, and log both application and process exceptions. These are just some of the benefits of implementing logging in your application.

You can implement logging utilizing a simple file-based approach such as the Internet Information Services (IIS) web logs. Another possible approach to logging application events might be to log data to a database. You should determine the exact storage mechanism based on the needs to query the log data. If you have to query the logged data, often a database may prove to be more beneficial. However, if your logged data is rarely looked at but must be done to satisfy the requirements of your application, then a simple text file may be sufficient. In addition, you can utilize the Windows event log features to log application events.

Other possible factors in determining the approach you use when logging application events can include the following:

- *Performance*: What is the minimum number of log entries that must be made within a given period of time?
- *Frequency*: At what interval will the log be added to?
- *Purging*: How long will it be before logged events can be purged out of the log files?
- *Readability*: Will you need to create a special application to view the log entries?
- *Scalability*: Does it make sense to contain the logs in a centralized location?

Another issue to consider is the configurability of the logging to be used in an application. For instance, should certain events be logged based on the environment that they're in? Should certain events be logged depending on whether the application is being debugged? Once you have determined the requirements of your application, you can then determine the best logging implementation approach.

Other Application Needs

When designing your application, you may find that your application has other needs as well. Some of these needs might include handling a disconnected environment, caching data, and encrypting and decrypting information, as well as dealing with application deployment.

Caching

Caching data can be useful in an application, especially when used to recall common lookup data often. Caching can provide the following benefits for an application:

- Improved application performance
- Improved scalability
- Reduced load on application and database servers

A perfect example of this is caching lookup data for drop-down list boxes in the user interface. In this case, an application would retrieve the lookup data on the initial load of the user interface and store that within the user interface so that it can be reused on subsequent loads for that particular user's UI component. This would save unnecessary hits to the database for that data that rarely or never changes.

When you cache data, you have to assess the data's volatility and determine how long you want to keep the cache around before expiring it and requiring the application to refresh the cache. Also, you must be aware of how much data is cached within a particular layer of the application, because caching can take up memory, and too much caching of data may unnecessarily take up too many resources from the running application. The use of caching must be balanced based on the amount of free local resources such as memory.

Cryptography

If you find that your application needs the ability to encrypt and decrypt data, you have to consider which methods will meet your needs and yet still perform well. The .NET Framework offers many options for handling the encryption and decryption of data; you can also create your own custom implementations or use third-party implementations if they meet your needs. The important thing you need to do is provide a consistent common interface for your cryptography needs in your application, thus allowing simplified maintenance and ensuring best-practice implementations.

Deployment

Another issue that you will more than likely have to address is the deployment of the application. For a web application, deployment is not as big of an issue as it would be for a Windows application. In a web application, you can simply deploy your application via Xcopy or an MSI to your production servers. However, a Windows application deployment can be a hair-raising issue, especially if you have many client workstations to which the application has to be deployed.

In many cases for large Windows application deployments, you must put a strategy in place for deploying the initial application and subsequent updates. The .NET Framework 2.0 had introduced a new technology called ClickOnce just for handling this kind of deployment.

■ **Note** ClickOnce is also present in the .NET Framework 3.0.

ClickOnce allows the deployment of applications over the Web. A user can click an application in the Start menu or use a link, and ClickOnce will determine based on a manifest whether the application should be updated. Although ClickOnce can handle some application deployment scenarios, it has some limitations. These limitations include the inability to modify registry settings, create custom installation directories, and share installations.

Another solution could be the use of a simple MSI installation package that a user can run themselves; however, even this scenario can have problems. Some of these problems can include the lack of rights, the user not performing the necessary updates, and installation problems such as prerequisite components not being present on the user’s machine.

Another possible solution is to create a custom bootstrapper application for downloading updates to a desktop. In many cases, you can buy third-party packages to do this, or you can find open source implementations on the Internet. Although this approach removes the need to push installation packages each time an application requires an update, an initial installation will still have to take place to get the bootstrapper on the user’s desktop.

Summary

In summary, this chapter has gone over the key components most applications will need in order to create successful, reliable, and scalable applications. Remember to break apart the components of an application in at least a logical manner. Having one class file that handles all the application functionality can become a nightmare to maintain; plus, having the components logically separated will allow for future growth by allowing for the physical separation of layers into components to facilitate scalability.

Now that you understand these components, the next task is to figure out how to implement these different layers and to understand how the Enterprise Library Application Blocks can fulfill these necessary features in an application.